CS744 (Autumn 2017)

Midsem Review: Practice Problems

1. Consider a clinic with one doctor and a very large waiting room (of infinite capacity). Any patient entering the clinic will wait in the waiting room until the doctor is free to see her. Similarly, the the doctor also waits for a patient to arrive to treat. All communication between the patients and the doctor happens via a shared memory buffer. Any of the several patient processes, or the doctor process can write to it. Once the patient "enters the doctors office", she conveys her symptoms to the doctor using a call to `consultDoctor()`, which updates the shared memory with the patient's symptoms. The doctor then calls `treatPatient()` to access the buffer and update it with details of the treatment. Finally, the patient process must call `noteTreatment()` to see the updated treatment details in the shared buffer, before leaving the doctor's office. A template code for the patient and doctor processes is shown below. Enhance this code to correctly synchronize between the patient and the doctor processes. Your code should ensure that no race conditions occur due to several patients overwriting the shared buffer concurrently. Similarly, you must ensure that the doctor accesses the buffer only when there is valid new patient information in it, and the patient sees the treatment only after the doctor has written it to the buffer. You must use **only semaphores** to solve this problem. Clearly list the semaphore variables you use and their initial values first. Please pick sensible names for your variables.

   (a) Semaphore variables and initial values:

   (b) Patient process:

```
consultDoctor();

noteTreatment();
```

   (c) Doctor process:

```
while(1) {

treatPatient();

}
```

2. Consider a producer-consumer situation, where a process P produces an integer using the function `produceNext()` and sends it to process C. Process C receives the integer from P and consumes it in the function `consumeNext()`. After consuming this integer, C must let P know, and P must produce the next integer only after learning that C has consumed the earlier one. Assume that P and C get a pointer to a shared memory segment of 8 bytes, that can store any two 4-byte integer-sized fields, as shown below. Both fields in the shared memory structure are zeroed out initially. P and C can read or write from it, just as they would with any other data object. Briefly describe how you would solve the producer-consumer problem described above, using *only* this shared memory as a means of communication and synchronization between processes P and C. You must not use any other synchronization or communication primitive. You are provided template code below which gets a pointer to the shared memory, and produces/consumes integers. You must write the code for communicating the integer between the processes using the shared memory, with synchronization logic as required.

```
struct shmem_structure {
int field1;
int field2;
};
```

(a) Producer:

```
struct shmem_structure *shptr = get_shared_memory_structure();

while(1) {
int produced = produceNext();




}
```

(b) Consumer:

```
struct shmem_structure *shptr = get_shared_memory_structure();

while(1) {
int consumed; //fill this value from producer


consumeNext(consumed);


}
```

3. Several concurrent threads/processes wish to read and write data shared between them. Some processes only want to read, while others want to update the shared data. Multiple readers may concurrently access the data. However, a writer must not access the data concurrently with anyone else, either a reader or a writer. Write down pseudocode to implement the functions readLock, readUnlock, writeLock, and writeUnlock that are invoked by the readers and writers to realize read/write locks. Write two versions of the solution: one using locks/conditional variables, and the other using only semaphores.

4. Consider a multithreaded banking application. The main process receives requests to tranfer money from one account to the other, and each request is handled by a separate worker thread in the application. All threads access shared data of all user bank accounts. Bank accounts are represented by a unique integer account number, a balance, and a lock of type `mylock` (much like a `pthreads` mutex) as shown below.

```
struct account {
int accountnum;
int balance;
mylock lock;
};
```

Each thread that receives a transfer request must implement the transfer function shown below, which transfers money from one account to the other. Add correct locking (by calling the `dolock(&lock)` and `unlock(&lock)` functions on a `mylock` variable) to the tranfer function below, so that no race conditions occur when several worker threads concurrently perform transfers. Note that you must use the fine-grained per account lock provided as part of the account object itself, and not a global lock of your own. Also make sure your solution is deadlock free, when multiple threads access the same pair of accounts concurrently.

```
void transfer(struct account *from, struct account *to, int amount) {




from->balance -= amount; // dont write anything...
to->balance += amount; // ...between these two lines


}
```

5. Consider a user level threading library that multiplexes $N > 1$ user level threads over $M \geq 1$ kernel threads. The library manages the concurrent scheduling of the multiple user threads that map to the same kernel thread internally, and the programmer using the library has no visibility or control on this scheduling or on the mapping between user threads and kernel threads. The N user level threads all access and update a shared data structure. When (or, under what conditions) should the user level threads use mutexes to guarantee the consistency of the shared data structure?

Answer _____

A. Only if $M > 1$.

B. Only if $N \geq M$.

C. Only if the M kernel threads can run in parallel on a multi-core machine.

D. User level threads should always use mutexes to protect shared data.

6. Consider a parent process P that has forked a child process C in the program below.

```
int a = 5;
int fd = open(...) //opening a file
int ret = fork();
if(ret >0) {
close(fd);
a = 6;
...
}
else if(ret==0) {
printf("a=%d\n", a);
read(fd, something);
}
```

After the new process is forked, suppose that the parent process is scheduled first, before the child process. Also, the OS implements copy-on-write during fork. Once the parent resumes after fork, it closes the file descriptor and changes the value of a variable as shown above. Assume that the child process is scheduled for the first time only after the parent completes these two changes.

(a) What is the value of the variable `a` as printed in the child process, when it is scheduled next? Explain.

(b) Will the attempt to read from the file descriptor succeed in the child? Explain.

7. Consider a multithreaded webserver running on a machine with N parallel CPU cores. The server has M worker threads. Every incoming request is put in a request queue, and served by one of the free worker threads. The server is fully saturated and has a certain throughput at saturation. Under which circumstances will increasing M lead to an increase in the saturation throughput of the server?

8. Consider a system with paging-based memory management, whose architecture allows for a 4GB virtual address space for processes. The size of logical pages and physical frames is 4KB. The system has 8GB of physical RAM. The system allows a maximum of 1K processes to run concurrently and uses a 10-bit number to represent the PID. (Assume 1K = 1024.) Assuming the OS uses hierarchical paging, calculate the maximum memory space required to store the page tables of *all* processes in the system. Assume that each page table entry requires an additional 10 bits (beyond the frame number) to store various flags. Assume page table entries are rounded up to the nearest byte. Consider the memory required for both outer and inner page tables in your calculations.

9. Consider a 4-way assocaitive cache of size 4MB, and 64-byte cache lines. Assume that the system uses 32-bit addresses. How many bits of the address will form the tag in the cache?

10. Consider a closed system, where every incoming user/job spends time (2 seconds and 6 seconds) at two components of the server, thinks for a time of Z seconds, before returning back into the system. It is found that a minimum of 10 concurrent users are required to saturate the system. What is the value of Z?